Atomic RMI – User Guide

Release 2.1

Wojciech Mruczkiewicz, Konrad Siek,
Paweł T. Wojciechowski
www.it-soa.pl/jpaxos

January 2011

Contents

1 Overview **3**

2 Compilation and deployment **5**
2.1 Deployment requirements . . . 5
2.2 Building the library . . . 5
2.3 Adding the precompiler . . . 6

3 Application programming interface **9**
3.1 Components . . . 9
3.2 RMI registry . . . 10
3.3 Remote objects . . . 10
3.4 Atomic transactions . . . 11
3.5 Lock initialization . . . 12
3.6 Exceptions . . . 12

4 In-depth look **15**
4.1 Accesses of transactional remote objects . . . 15
4.2 Distributed transactions . . . 15
4.3 Multi-threaded transactions . . . 17
4.4 Nested transactions . . . 18
4.5 Recurrency . . . 18
4.6 Failures . . . 18

5 Complete example: the bank **21**
5.1 Remote objects . . . 21
5.2 Server . . . 22
5.3 Audit client . . . 22
5.4 Transfer client . . . 23
5.5 Transfer client with retry . . . 24
5.6 Running the example . . . 25

6 Automatic inference of accesses **27**
6.1 Running the precompiler . . . 27
6.2 Commandline options . . . 28

6.3 Manual override 28

7 Papers and reports **31**

Contents:

CHAPTER 1

Overview

Atomic RMI is an extension of Java Remote Method Invocation with distributed atomic transactions.

Java Remote Method Invocation (Java RMI) is a system for creating distributed Java technology-based applications, where methods of remote Java objects may be invoked from other *Java Virtual Machines (JVMs)* on the same host or on different hosts. Java RMI marshals and unmarshals parameters of remotely-called methods retaining object-oriented polymorphism and type information.

Our library provides constructs on top of Java RMI allowing the programmer to declare a series of method calls on remote objects as a distributed transaction. Such a transaction guarantees the ACID properties of *atomicity* (either all of the operations of a transaction are performed or none), *consistency* (after any transaction finishes, the system remain in a valid—or consistent—state), and *isolation* (each transaction perceives itself as being the only currently running transaction). *Durability* (when a transaction is completed its effects are permament) is not normally supported by *Software Transactional Memory (STM)* in general, and it is also not supported by Atomic RMI.

Atomic RMI exercises pessimistic concurrency control using fine grained locks (a single lock per remote object) while simultaneously providing support for rolling back transactions (using a `rollback` construct), and restarting them (using a `retry` construct).

CHAPTER 2

Compilation and deployment

A quick overview on what is necessary to compile the library and how to start the example which is included in the distribution.

2.1 Deployment requirements

Atomic RMI requires the following components to be installed on the target system:

- Java SDK 1.5 or later
- Code Generation Library (cglib) 2.2 (distributed under the Apache License 2.0)

Additionally, the following are optional but helpful tools in compilation and packaging:

- Apache Ant — used to run all build scripts (distributed under the Apache License 2.0)
- jdeb 0.7 or later — used to allow Ant to create debian packages (distributed under the Apache License 2.0)

2.2 Building the library

Atomic RMI can be compiled using Apache Ant, a popular Java-based build tool. Prior to building, the `build.properties` configuration file can be adjusted:

- `cglib.file` — name of the file containing the Code Generation Library
- `cglib.version` — version of the Code Generation Library
- `jdeb.file` — name of the file containing the jdeb library
- `dir.lib` — directory containing all external libraries
- `dir.src` — directory containing source code
- `dir.test` — directory containing source code of tests and examples
- `dir.doc` — directory containing the documentation
- `dir.build` — directory where Java binaries (class files) will be generated
- `dir.dist` — directory where distribution packages (`deb`, `rpm`, `jar`) will be generated

- `dir.javadoc` — directory where Javadoc documentation will be generated
- `dir.deb` — temporary directory used to create a `deb` package
- `dir.rpm` — temporary directory used to create an `rpm` package
- `javadoc.title.window` — title of all Javadoc pages
- `javadoc.title.document` — title of the main Javadoc document
- `javadoc.bottom` — footer of all Javadoc documents
- `dir.system.jar` — path for deploying generated `jar` files on Unix systems.
- `dir.system.doc` — path for deploying documentation on Unix systems.
- `example.host` — hostname used in the bank example
- `example.port` — port used in the bank example

A number of typical Ant targets are available:

- `all` — run all build targets
- `compile` — compile the library, tests, and examples
- `dist` — create all distribution packages
- `jar` — create a Java archive from compiled sources
- `deb` — create a Debian package from compiled sources
- `rpm` — create a Red Hat package from compiled sources
- `tar` — create a tarball from source code
- `javadoc` — generate Javadoc API documentation
- `html` — generate HTML documentation using Sphinx
- `clean` — remove all generated files

Alternatively, targets for building Atomic RMI and the precompiler simultaneously may be used, as described in *chap-atomicrmi-compilation-precompiler*.

Running the `ant` command in the project's directory with no arguments or with the `all` target will create a distribution-ready set of packages:

```
cd <project-directory>
ant all
```

A successful creation of distribution packages can be given a crash test using the example provided in the source code (described in *Complete example: the bank*.)

2.3 Adding the precompiler

To use the tool for automatic inference of transactional remote object access information (described in *Automatic inference of accesses*), the following additional considerations need to be taken when deploying Atomic RMI.

The precompiler requires the following components to be installed on the target system:

- Java SDK 1.5 or later

- Soot 2.4.0 or later (distributed under the GNU Lesser General Public License 2.1)
- Jasmin 2.4.0 or later (required by Soot, distributed under the GNU General Public License)
- Polyglot 2.4.0 or later (distributed under the GNU Lesser General Public License 2.1 and the Eclipse Public License 1.0)

The `precompiler.properties` configuration file may need to be adjusted prior to building:

- `dir.precompiler.src` — directory containing source code of the precompiler
- `soot.file` — name of the file containing the soot framework
- `jasmin.file` — name of the file containing the jasmin library
- `polyglot.file` — name of the file containing the polyglot library

Afterwards, the precompiler can be built using Atomic RMI's Ant build script. It is recommended to use the following targets, which will jointly prepare the precompiler and Atomic RMI:

- `all-all` — create distribution packages for Atomic RMI and the precompiler
- `compile-all` — compile Atomic RMI with precompiler support
- `dist-all` — create distribution packages Atomic RMI with precompiler support
- `jar-all` — create a Java archive from compiled sources
- `tar-all` — create a tarball from compiled sources
- `javadoc-all` — generate Javadoc API documentation for Atomic RMI and the precompiler

Alternatively, the following targets are precompiler-specific (some existing files may be overwriten):

- `all-precompiler` — create distribution packages for the precompiler
- `compile-precompiler` — compile the precompiler
- `dist-precompiler` — create distribution packages for the precompiler
- `jar-precompiler` — create a Java archive from compiled sources
- `tar-precompiler` — create a tarball from compiled sources
- `javadoc-precompiler` — generate Javadoc API documentation for the precompiler

Running the `ant` command in the project's directory with the `precompiler-all` target will create a ready-to-use executable jar file:

```
# cd <project-directory>
# ant precompiler-all
```

CHAPTER 3

Application programming interface

A detailed overview of the library's API elements and how to use them to create a working distributed transactional system.

3.1 Components

A typical system using Java RMI consists of a number of JVMs on one or more hosts. A number of remote objects are created on any of those machines and registered in an RMI registry located on the same host. A client running on any JVM will access those remote objects, having first located them via the RMI registry. Proxies provide rollback capabilities and control method invocations to ensure that the properties of atomicity, consistency, and isolation are followed.

The components of the system are shown in fig-atomicrmi-architecture, and can be related to the components of a system using Java RMI but not Atomic RMI in fig-atomicrmi-rmi.

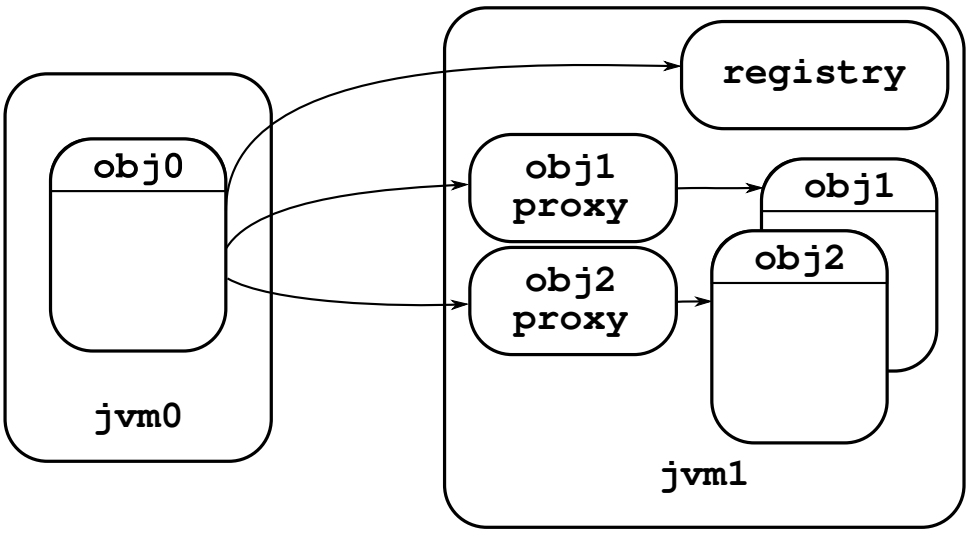

Figure 3.1: The components of a system using Atomic RMI

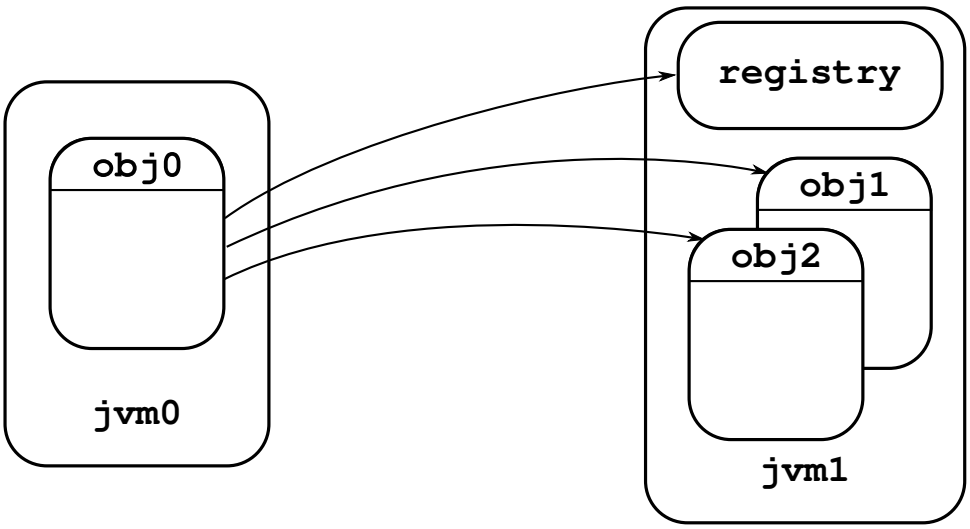

Figure 3.2: The components of a system using only Java RMI

3.2 RMI registry

The Atomic RMI library uses the RMI registry both to distribute and to locate remote objects. Typically, this means using the default implementation of the interface `Registry` from the `java.rmi.registry` package, although other implementations of that interface can be used just as well.

The registry is an external service that runs on a specific host computer and listens to a particular port (`1099` by default). The programmer can gain access to the registry by using the static `getRegistry(host, port)` method of class `LocateRegistry` from the `java.rmi.registry`. Once the RMI registry is obtained, the various remote objects can be registered by the server, or located by the client.

3.3 Remote objects

Remote objects are the shared resources of a distributed system using the Atomic RMI library. Remote objects are created by the programmer with very few restrictions.

All remote objects should implement an interface created by the programmer, which extends the `java.rmi.Remote` interface, for example:

```
interface MyRemote extends Remote {
    void doSomething() throws RemoteException;
}
```

When extending `java.rmi.Remote` all the methods of the interface should declare to throw `java.rmi.RemoteException`. This mechanism is used by the underlying Java RMI framework to move objects from server to server and direct method invocations.

The following code illustrates how the interface should be implemented:

```
class MyRemoteImpl extends TransactionalUnicastRemoteObject implements MyRemote {
    public void doSomething() throws RemoteException {
        ...
    }
}
```

It is important to note that all remote objects that are a part of transactional executions need to extend the class `soa.atomicrmi.TransactionalUnicastRemoteObject`, which acts as a wrapper and extends the remote object implementation with counters used by the concurrency control algorithms, and the ability to create checkpoints to which the objects can be rolled back.

Objects created from such devised remote object classes must be registered (bound) with the RMI registry on the server side. This is done with the use of either the `bind(name, object)` method or the `rebind(name, object)` method of the `Registry` instance. Then, the objects may be instantiated on the client side using the `lookup(name)` method of the registry. For example:

```
Registry registry = LocateRegistry.getRegistry("localhost");
MyRemote obj = new MyRemoteImpl();
registry.rebind("B", b);
```

3.4 Atomic transactions

Distributed atomic transactions are controlled with the use of instances of the `Transaction` class from the `soa.atomicrmi` package. A transaction object first needs to be initialized with the constructor, then its preamble must be defined. Finally, the transaction is started with the method `start` and ended either with the method `commit`, `rollback`, or `retry` (the latter requires using the `Transactable` interface described later on). Between the two methods the invocations of remote objects are traced and delayed if necessary. This guarantees sequential execution of atomic transactions and guarantees the atomicity constraints.

The following code shows a fully defined transaction:

```
Transaction transaction = new Transaction(registry);
object = (MyRemote) transaction.accesses(object, 1);
transaction.start();
object.doSomething();
transaction.commit(); // or: transaction.rollback();
```

The transaction preamble provides information about object accesses which is necessary for the dynamic scheduling of method calls to remote objects. The preamble is constructed by calling the method `accesses(object, calls)` on the instance of the transaction for each remote object used in the transaction: the object reference is passed as the first argument, and the second argument is an upper bound (supremum) on the number of times the indicated object is used within the transaction. The method returns an object wrapped by special object proxy. During transaction execution only this proxy must be used to guarantee atomicity and isolation properties.

The supremum number of invocations of each object—-the second argument of the `accesses(object, calls)` method—-may be collected manually or inferred automatically by the precompiler (described in *Automatic inference of accesses*). In case that the exact number is unknown, an upper bound may be given or the number may be omitted altogether, keeping in mind, that the more relaxed the bounds, the fewer transactions may be executed in parallel (although the guarantees of atomicity and isolation are still not violated). If the given number of maximum method calls is lower than the actual number of calls, the guarantees provided by the library could not be upheld, and so a `VersionedRMIException` exception is raised.

An alternative way of creating a transaction is to use the `Transactable` interface from the package `soa.atomicrmi`, in the following manner:

```
Transaction transaction = new Transaction(registry);
object = (MyRemote) transaction.accesses(object, 1);
```

```
transaction.start(new Transactable() {
    public void atomic(Transaction t) throws RemoteException {
        object.doSomething();
        t.retry();
    }
});
```

The programmer implements the `Transactable` interface (either by instantiating an object of an anonymous class or by creating a new class) and overloads the method `atomic(transaction)` using the code that would normally be inserted between the transaction's *start* and *commit*, *rollback*, or *retry*, with the exception that `commit`, `rollback`, and `retry` are called on the transaction object passed via the method's argument. An instance of a class implementing the `Transactable` interface is then passed as an argument to the `start(transactable)` method of the transaction object. It is obligatory to use this way of defining transactions to use the retry mechanism.

3.5 Lock initialization

Starting transactions (executing a transaction's `start` method) may only happen one-at-a-time within the entire system, because of the operations on counters of various objects that need to be performed at the time. As a result of this, a global lock is used at that point. The lock must be initialized on the server side using the `initialize(registry)` static method of the class `soa.atomicrmi.TransactionsLock`.

It is important to note that this lock is only held for the short time required to start transactions and does not play further part in guaranteeing atomicity, consistency, nor isolation.

```
public class Server {
    public static void main(String[] args) {

        ...

        // Initialize synchronization mechanisms for transactions.
        TransactionsLock.initialize(registry);
    }
}
```

3.6 Exceptions

A number of exceptions are used within Atomic RMI to alert of unexpected issues:

- `RetryCalledException` — thrown and propagated upwards when a retry operation is requested outside of a `Transactable`-type object,
- `RollbackForcedException` — thrown when a rollback is required during remote method invocation or while committing a transaction, and performing the invocation or commit would cause an inconsistent state to develop,
- `TransactionException` — thrown when any of a number of serious problems occur during the execution transaction. In particular, this means that one of the following actions had caused an erroneous situation to occur:
  - *on providing object access information* — an invalid upper bound on number of remote object method invocations was given,

- *on providing object access information* — providing object access information was attempted while the transaction was in a wrong state (e.g., already running),
- *on providing object access information* — creation of a proxy for a remote object failed (due to an exception on the remote host),
- *on starting transaction* — transaction could not be started due to a failure on the remote host even though several attempts were made,
- *on starting transaction* — transaction could not be initialized because the transaction tried to transition from a wrong initial state or to a wrong target state,
- *on starting transaction* — transaction could not be initialized because of an exception on the remote host,
- *on invoking a remote method* — the specified upper bound is lower than the actual number of invocations,
- *on invoking a remote method* — a snapshot of the remote object could not be created due an input/output problem on the remote host,
- *on invoking a remote method* — a lock to a version counter could not be obtained,
- *on invoking a remote method* — object could not be freed because of an exception on the remote host,
- *on finishing a transaction* — a remote object could not be re-created from a snapshot due to supplying an inappropriate object type or to an input/output problem on the remote host,
- *on finishing a transaction* — a lock to a version counter could not be obtained,
- *on finishing a transaction* — transaction could not be initialized because the transaction tried to transition from a wrong initial state or to a wrong target state,
- *on freeing a remote object* — the object that is to be freed was not specified in the preamble, and so this transaction has no access information regarding that object,
- *on finishing a transaction* — object could not be freed because of an exception on the remote host,
- *on finishing a transaction* — releasing a transaction lock was attempted while the transaction was in a wrong state (e.g., not yet running),
- *on finishing a transaction* — releasing a transaction lock was attempted while the transaction was not the owner of the lock,
- *on obtaining the global lock* — the transaction lock is inaccessible because or an exception on the remote host,

CHAPTER 4

In-depth look

A guide to some basic conepts behind Atomic RMI which affect the end user.

4.1 Accesses of transactional remote objects

It is recommended that an Atomic RMI user provides information about how many times, at maximum, each transactional remote object is invoked: this information is used to control the way in which remote objects are accessed by all the transactions in the system.

The maximum number of invocations of each object must either be collected manually or inferred automatically by the precompiler (described in *Automatic inference of accesses*) for each transaction. It is prefered that the predicted number of remote object invocations is identical with their actual number. But if the exact number is unknown, an upper bound may be given or the number may be omitted altogether, keeping in mind, that the more relaxed the bounds, the fewer transactions may be executed in parallel (although the guarantees of atomicity and isolation are still not violated). It is essential that the number of maximum method calls is never lower than the actual number of calls, because then the guarantees provided by the library could not be upheld. If this occurs, `VersionedRMIException` exception is thrown.

4.2 Distributed transactions

Atomic RMI may be used to create distributed transactions, such where a transaction started on one host calls methods of objects located on another host, and those methods involve invoking methods of other remote objects (as shown in fig-atomicrmi-distributed).

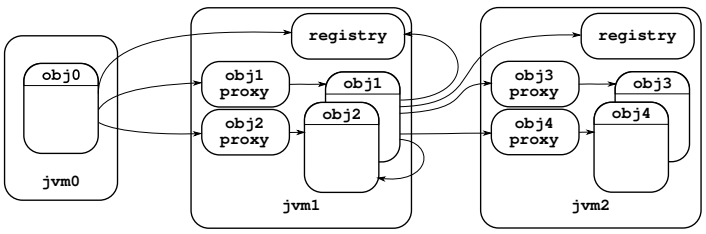

Figure 4.1: Distributed transactions: `obj1` accesses `obj2`, `obj3` and `obj4`

When creating distributed transactions it is also necessary for the programmer to declare maximum accesses to all remote objects, even those used within other remote objects' methods. Atomic RMI provides no additional mechanisms to facilitate this, but a simple addition to the remote objects' interface can be used to mitigate the inconvenience.

First, the programmer may declare a method that returns all the other remote objects that a given remote object uses. For simple cases just the reference to an object can be returned, while a more complex case can return a collection indicating both which objects are used and up to how many times. The following example illustrates an interface which declared two methods, `a` and `b`, which do something that uses some remote objects, and two more methods, `getObjectForA` and `getAccessesForB` which return information about remote object accesses performed in `a` and `b`.

```
interface DistributedRemote extends Remote {
    void a();
    void b();
    Remote getObjectForA();
    Map<Remote, Integer> getAccessesForB();
}
```

When a transaction is declared using `DistributedRemote`-type objects, the accesses will include information about how many times those object may be used at maximum, and also at-most how many times the objects used within their methods may be used. For example:

```
Transaction t = new Transaction(registry);

DistributedRemote r = (DistributedRemote) registry.lookup("A");

// Which remote objects are used from this host and how many times?
r = (DistributedRemote) t.accesses(r, 2);

// Variant 1: which remote object does r use whenever we call the method a?
t.accesses(r.getObjectForA());

// Variant 2: which remote objects does r use whenever we call the method b
// and how many times?
Map<Remote, Integer> accessesForB = t.getAccessesForB();
for (Remote object : accessesForB.keySet()) {
    Integer upperBound = accessesForB.get(object);
    t.accesses(object, upperBound);
}

t.start();

r.a();
r.b();

t.commit();
```

An example implementation of that interface is presented below:

```
class DistributedRemoteImpl extends TransactionalUnicastRemoteObject implements DistributedRemote {

    private MyRemote o1, o2, o3;

    DistributedRemoteImpl(MyRemote o1, MyRemote o2, MyRemote o3) {
        this.o1 = o1;
        this.o2 = o2;
        this.o3 = o3;
```

```
    }

    public void a() {
        o1.doSomething();
    }

    public void b() {
        o1.doSomething();
        o2.doSomething();
        o3.doSomething();
    }

    public Remote getObjectForA() {
        return o1;
    }

    public Map<Remote, Integer> getAccessesForB() {
        Map<Remote, Integer> map = new HashMap<Remote, Integer>();
        map.put(o1, 1);
        map.put(o2, 1);
        map.put(o3, 1);
        return map;
    }
}
```

4.3 Multi-threaded transactions

Atomic RMI does not make any allowances for threads started within transactional code. A multi-threaded transaction may be created, but all matters of synchronization are then left to the programmer (in other words, Atomic RMI does not guarantee isolation of the various threads in a multi-threaded transaction). The programmer should therefore consider:

- making certain that no thread invokes the `commit`, `rollback`, or `retry` methods after another thread had done so, which will confuse the state of the transaction, and cause an exception to be raised;
- making certain that no thread tries to access any transactional remote objects after any thread invokes a `commit`, `rollback`, or `retry` method, which may leave the system in an incoherent state or cause other unforseen problems;
- ensuring that the maximum number of object accesses is properly declared no matter how the operation within the threads are interwoven, because a declared number of accesses lower than their actual number, the guarantees described in *Overview* may be violated and an exception will be raised.

In addition to the problems mentioned above, it is necessary for the programmer to create any synchronization mechanisms that may be required, as Atomic RMI provides none for threads within a single transaction.

Note that running transactions in a separate thread while the entire transaction is completely within that thread causes no issues to arise.

4.4 Nested transactions

Although it is not recommended, it is possible for an Atomic RMI user to nest transactions within other transactions. In such cases it is vital for the programmer to ensure that an inner transaction does not use any of the transactional objects used by outer transactions. If this condition is not ensured, Atomic RMI will cause a deadlock.

4.5 Recurrency

Atomic RMI allows recurrent starting of the same transactional code, provided the following conditions are met:

- the transaction is defined using the `Transactable` interface (as described in: *Application programming interface* (see sec. 3)),
- the maximum possible number of accesses of remote objects' methods in recurring invocations are accounted for in the transactions preamble,

Then, the programmer can simply call the method `atomic` again within itself to create recursion.

The execution will proceed until the methods `commit`, `rollback`, or `retry` are called, in which case the `atomic` method is exited and the transaction finishes as normal.

In case when the programmer does not use the `Transactable` interface to define a transaction, calling the method `start` multiple times will not result in recursion, but instead an exception will be raised at run-time.

4.6 Failures

Atomic RMI can suffer two basic types of failures: failures of remote objects and failures of transactions.

4.6.1 Failures of remote objects

Failures of remote objects are straightforward and the responsibility for detecting them and alarming Atomic RMI falls onto the mechanisms built into Java RMI. Whenever a remote object is called from a transaction and it cannot be reached, it is assumed that this object has suffered a failure and as a result a `RemoteException` is thrown at run-time. The programmer may then choose to handle that exception by, for example, rolling the transaction back, re-running it, or compensating for the failure.

Failures of remote objects follow a *crash-stop* model, where an object that has crashed is not brought back to operation, but simply removed from the system.

4.6.2 Failures of transactions

Failures of transactions are such failures where remote objects that are operated on by a transaction loose communuication to that transaction and the transaction is considered to have crashed. When such a failure is detected the affected remote objects revert to the state immediately prior to the start of the transaction, so that all changes done by the failed transaction are forgotten.

Failure detection is performed as follows: transactions send a heartbeat signal to all transactional remote objects they use, and when a remote object stops receiving this signal (a connection times out) it assumes that a failure has occured. This means, that if the heartbeat signal is significantly delayed the transaction may continue running while one or more remote objects it uses are under the impression that it has failed.

However, the implementation of failure handling in Atomic RMI assumes a perfect failure detector, which ensures that all the entities involved in a failure (all the transactional remote objects and the failed transaction itself) agree whether the transaction has actually crashed or is still running. Therefore, it is possible for Atomic RMI to lose consistency if the failure detector is wrong in detecting a failure.

CHAPTER 5

Complete example: the bank

The following describes an example showing how to create a simple distributed application using the Atomic RMI library.

The example is included with the Atomic distribution RMI and can be run using pre-prepared Ant targets.

5.1 Remote objects

The example includes a single type of remote object and it is specified by the interface `Account`. It provides two methods: `getBalance` and `setBalance` for determining and setting the bank account balance.

```
public interface Account extends Remote {
    public int getBalance() throws RemoteException;
    public void setBalance(int balance) throws RemoteException;
}
```

A class implementing that interface is presented below. The method `getBalance` gives the value of the internal field `balance`. The `setBalance` method assigns a new value to this field.

```
public class AccountImpl extends TransactionalUnicastRemoteObject implements Account {
    protected int balance;
    public AccountImpl(int balance) throws RemoteException {
        this.balance = balance;
    }
    public int getBalance() throws RemoteException {
        return balance;
    }
    public void setBalance(int balance) throws RemoteException {
        this.balance = balance;
    }
}
```

The class `AccountImpl` must extend the `TransactionalUnicastRemoteObject` class from the `soa.atomicrmi` package to allow this remote object to be available remotely and fitted with the appropriate transactional mechanisms. The standard Java RMI library also allows to use the static `ExportObject` method from the `UnicastRemoteObject` class (in such case deriving from class `UnicastRemoteObject` is no longer required). This mechanism is not supported by the Atomic RMI and only the first option can be used.

5.2 Server

Generally the server implementation should include the following steps:

- Reference to the `Registry` must be obtained to allow binding remote objects;
- Remote objects must be instantiated, given identifiers, and registered using with the `Registry` object;
- `TransactionsLock` must be initialized.

The following server implementation performs those steps in order to create two bank accounts. The first is initialized with the balance of 1000 and registered as "A". The second is initialized with the balance of 500 and registered as "B".

```
public class Server {
    public static void main(String[] args) throws RemoteException {
        // Get a reference to RMI registry.
        Registry registry = LocateRegistry.getRegistry("192.168.1.10", 1099);

        // Initialize bank accounts.
        Account a = new AccountImpl(1000);
        Account b = new AccountImpl(500);

        // Bind addresses.
        registry.rebind("A", a);
        registry.rebind("B", b);

        // Initialize synchronization mechanisms for transactions.
        TransactionsLock.initialize(registry);
    }
}
```

5.3 Audit client

Two clients are used to show the usage of atomic transactions. This client retrieves the balance of accounts A and B, and prints the total balance of those two accounts. Balance retrieval is done within the atomic transaction.

To implement those clients these general steps should be followed:

- A reference to the `Registry` must be located.
- Remote object references must be located with the use of the `lookup` method of the `Registry` instance.
- New instance of `Transaction` must be instantiated.
- The transaction preamble must be described using the `accesses` method of the `Transaction` object and wrapping remote objects code that will transparently control the way those objects will be used. (We use new variables, `ta` and `tb` for wrapped objects here, but these objects may also be assigned to the old variables `a` and `b` with which they share a common interface.)
- Atomic transaction execution must be contained between the `start` method and any of the `commit` or `rollback` methods of the instance of `Transaction`.

The code below implements the first client that is responsible for retrieving the total balance. In the atomic transaction each of the remote objects is accessed exactly once and this value is described in the preamble before the transaction begins. The balance of accounts A and B is retrieved within the transaction.

```
public class AuditClient {
    public static void main(String[] args) throws RemoteException, NotBoundException {
        // Get a reference to RMI registry.
        Registry registry = LocateRegistry.getRegistry("192.168.1.10", 1099);

        // Get references to remote objects.
        Account a = (Account) registry.lookup("A");
        Account b = (Account) registry.lookup("B");

        // Transaction preamble.
        Transaction transaction = new Transaction(registry);
        Account ta = (Account) transaction.accesses(a, 1);
        Account tb = (Account) transaction.accesses(b, 1);

        transaction.start();

        // Check balance on both accounts atomically.
        int balanceA = ta.getBalance();
        int balanceB = tb.getBalance();

        transaction.commit();

        System.out.println(balanceA + balanceB);
    }
}
```

When running multiple clients simultaneously from various hosts, using atomic transactions guarantees that no transfer can be interleaved with any other transfer or balance retrieval operations, so the total balance is always constant.

5.4 Transfer client

The second client transfers money from account A to account B and commits or rolls back. This transfer is also done within the atomic transaction. The implementation of the second client is quite similar:

- A reference to the `Registry` is located.
- Remote object references are located with the use of the `lookup` method of the `Registry` instance.
- New instance of `Transaction` is instantiated.
- The transaction preamble is described using the `accesses` method of the `Transaction` object and the remote objects are recreated.
- Atomic transaction execution is contained between the `start` method and any of the `commit` or `rollback` methods of the instance of `Transaction`.

This time there are two accesses to remote objects A and B, and this is accounted for in the task description. Additionally, the transaction can finish with either a commit, or rollback, depending on some external confirmation.

```
public class TransferClient {
    public static void main(String[] args) throws RemoteException, NotBoundException {
        // Get a reference to RMI registry.
        Registry registry = LocateRegistry.getRegistry("192.168.1.10", 1099);

        // Get references to remote objects.
        Account a = (Account) registry.lookup("A");
        Account b = (Account) registry.lookup("B");

        // Transaction header.
        Transaction transaction = new Transaction(registry);
        a = (Account) transaction.accesses(a, 2);
        b = (Account) transaction.accesses(b, 2);

        transaction.start();

        // Retrieve balance on both accounts.
        int balanceA = a.getBalance();
        int balanceB = b.getBalance();

        // Transfer funds from A to B.
        a.setBalance(balanceA - 100);
        b.setBalance(balanceB + 100);

        // End transaction.
        if (confirm()) {
            transaction.commit();
        } else {
            transaction.rollback();
        }
    }
}
```

5.5 Transfer client with retry

This client is functionally the same as the other transfer client, except that it gives the option to retry the transaction. In order to achieve that, it must use the `Transactable` interface to define the transaction:

- A reference to the `Registry` is located.
- Remote object references are located with the use of the `lookup` method of the `Registry` instance.
- New instance of `Transaction` is instantiated.
- The transaction preamble is described using the `accesses` method of the `Transaction` object and the remote objects are recreated.
- An object of a class implementing the `Transactable` interface is created containing the transaction, which is concluded by any of the `commit`, `retry`, or `rollback` methods of the instance of `Transaction`. (For brevity, we create an anonymous class in the example.)
- Atomic transaction execution is commenced when the `start` method is called with the the `Transactable` instance as an argument.

Apart from the possibility of retrying instead of rolling back, the transaction is identical to that in the Transfer client without retry.

```
public class TransferClient {
    public static void main(String[] args) throws RemoteException, NotBoundException {
              // Get a reference to RMI registry.
        Registry registry = LocateRegistry.getRegistry("192.168.1.10", 1099);

        // Get references to remote objects.
        Account a = (Account) registry.lookup("A");
        Account b = (Account) registry.lookup("B");

        // Transaction header.
        Transaction transaction = new Transaction(registry);
        a = (Account) transaction.accesses(a, 2);
        b = (Account) transaction.accesses(b, 2);

        transaction.start(new Transactable() {
            public void atomic(Transaction t) throws RemoteException {
                // Retrieve balance on both accounts.
                int balanceA = a.getBalance();
                int balanceB = b.getBalance();

                // Transfer funds from A to B.
                a.setBalance(balanceA - 100);
                b.setBalance(balanceB + 100);

                // End transaction.
                if (confirm()) {
                    t.commit();
                } else {
                    t.retry();
                }
            }
        });
    }
}
```

5.6 Running the example

The example is included with the Atomic RMI distribution (extended to wait for a user's key press before committing or retrying a transaction). Scripts to run the example where included in the Apache Ant build file.

Before running the example, the following conditions must be met:

- the code of the library must be compiled
- the RMI registry must be listening on port 1099 on the same host as the server (the host and port may be set using the `build.properties` file as describes in Getting started.

Then, the Ant scripts for each application may be executed for the server:

```
rmiregistry 1099 &                  #start in background
cd <project-directory>
ant start-example-server
```

To run the client described in Audit client:

```
ant start-example-client-audit
```

To run the client described in Transfer client:

```
ant start-example-client-transfer
```

To run the client described in Transfer client with retry:

```
ant start-example-client-transfer-retry
```

CHAPTER 6

Automatic inference of accesses

Atomic RMI Precompiler is a tool which serves to help the users of Atomic RMI by automatically inferring upper bounds on the number of times each transactional remote object may be used in each transaction.

It is a commandline utility which analyzes Java source files (or complete projects) and, on the basis of the information collected during the analysis, generates some additional lines of code. This code specifies for each transaction which objects may be used within that transaction and up to how many times each of them may be expected to be invoked. These instructions are then inserted into the source code (either in-place, or into new files) before each transaction begins.

Before the precompiler is ready to use, it must be built. Detailed instructions for preparing the precompiler for use are given in *Adding the precompiler* (see sec. 2.3).

6.1 Running the precompiler

The precompiler can be run using one of the scripts provided with the distribution (appropriate to the operating system. For Unix-like operating systems:

```
./precompiler.sh --classpath <CLASSPATH> --sources <SOURCES>
```

For Windows operating systems:

```
precompiler.bat --classpath <CLASSPATH> --sources <SOURCES>
```

The two commandline options specified here are important to the analysis, and should be used as follows:

`sources` Specifies a path to a source code directory which will be used for analysis. The user can specify any number of directory paths this way, and they will be used in conjunction. This option is not optional: at least one definition of `sources` is necessary.

`classpath` Specifies a classpath for the source defined in `sources`. This needs to be defined so that the analyzer has complete information about anything referrenced within the analyzed source code. The classpath is specified once as a list of directories and archive files separated with a colon (`:`, on Unix-like systems) or a semi-colon (`;`, on Windows systems). For example:

```
SomeProject/bin:SomeProject/lib/some-lib.jar:.

    SomeProject\bin;SomeProject\lib\some-lib.jar;.
```

Both scripts assume that the JAVA_HOME global system variable is properly set to the directory where Java is installed.

Additionally, it is possible to run the precompiler via the Java runtime, for instance:

```
JASMIN=$ATOMICRMI_HOME/lib/jasminclasses-2.4.0.jar
POLYGLOT=$ATOMICRMI_HOME/lib/polyglot-1.3.3.jar
SOOT=$ATOMICRMI_HOME/lib/sootclasses-2.4.0.jar
PRECOMPILER=$ATOMICRMI_HOME/dist/atomicrmi-precompiler-0.2.jar

export JAVA_HOME=/usr/lib/jvm/java-6-sun
export CLASSPATH=$SOOT:$JASMIN:$POLYGLOT:$PRECOMPILER

java -cp $CLASSPATH soa.atomicrmi.precompiler.Precompiler \
    --sources <SOURCES> --classpath <CLASSPATH>
```

6.2 Commandline options

The precompiler may be configured during run-time using the following commandline options:

--sources=DIR Specify a source directory for analysis: this can be repeated to specify multiple directories.

--classes=CLASSPATH Set the classpath for the analyzed source code.

--output=DIR Set the output directory for the generated Java source files. If no output directory is specified the original source files are overwritten.

--manual=TEXT Set the text of comments used to indicate manually-added remote object accesses information, the default is: `@manual`.

--graphs=DIR Set the directory where graphs will be generated (if no directory is specified, no graphs are created).

--blacklist=LIST Specify a comma-separated list of classes which will not be analyzed; the classes are given as fully-qualified names which can also include a trailing asterisk * as a wildcard. The default blacklist is: `java.rmi.*,java.lang.*,java.util.*,java.io.*,soa.atomicrmi.*`.

--whitelist=LIST Specify a comma-separated list of classes which will definitely be analyzed, even if they are included in the blacklist; the classes are given as fully-qualified names which can also include a trailing asterisk * as a wildcard. The default whitelist is: `java.lang.Enum`.

--help or -h Print usage information and quit.

6.3 Manual override

When the precompiler generates transactional remote object access information, any accesses defined in the code by the programmer will be removed and overwriten. However, the user may override the precompiler's estimation, if a line of code is annotated with a trailing comment containing the string `@manual`, as seen in the code example below.

```
Transaction transaction = new Transaction(registry);
object = (MyRemote) transaction.accesses(object, argv.length); // @manual
transaction.start()
```

```
for (int i = 0; i < argv.length; i++) {
    object.doSomething(argv[i]);
}

transaction.retry();
```

The `@manual` string may be changed to any other text at runtime using the appropriate commandline option described in *Commandline options* (see sec. 6.2).

CHAPTER 7

Papers and reports

- Atomic RMI 1.0: Documentation and user's manual. M. Mamoński, W. Mruczkiewicz, K. Siek, P. T. Wojciechowski. Technical Report TR-ITSOA-OB2-2-PR-09-??, Institute of Computing Science, Poznań University of Technology, December 2009.
- Atomic RMI: Projekt i implementacja rozszerzenia mechanizmu zdalnego wywołania metod o wsparcie dla operacji atomowych (In Polish). Mariusz Mamoński, Paweł T. Wojciechowski. Technical Report TR-ITSOA-OB2-2-PR-09-??, Institute of Computing Science, Poznań University of Technology, December 2009.
- Analiza i porównanie wybranych realizacji Software Transactional Memory (In Polish). Piotr Kryger, Paweł T. Wojciechowski. Technical Report TR-ITSOA-OB2-2-PR-09-2, Institute of Computing Science, Poznań University of Technology, May 2009.
- Analiza mechanizmów wsparcia dla operacji atomowych w systemach rozproszonych i opartych na paradygmacie SOA (In Polish). Paweł T. Wojciechowski. Technical Report TR-ITSOA-OB2-2-PR-09-1, Institute of Computing Science, Poznań University of Technology, February 2009.
- Extending Atomic Tasks to Distributed Atomic Tasks. Paweł T. Wojciechowski. In the Proc. of CAV '08 (the 20th International Conference on Computer Aided Verification) - Workshop on Exploiting Concurrency Efficiently and Correctly (EC)^2, July 2008.
- *Statically Computing Upper Bounds on Object Calls for Pessimistic Concurrency Control*. Konrad Siek and Pawel Wojciechowski. In the Proc. of FLoC '10 (the Federated Logic Conference) - Workshop on Exploiting Concurrency Efficiently and Correctly EC2 2010, July 2010.